

DEMYSTIFYING PS2 FORTUNA THROUGH OPENTUNA

by alexparrado

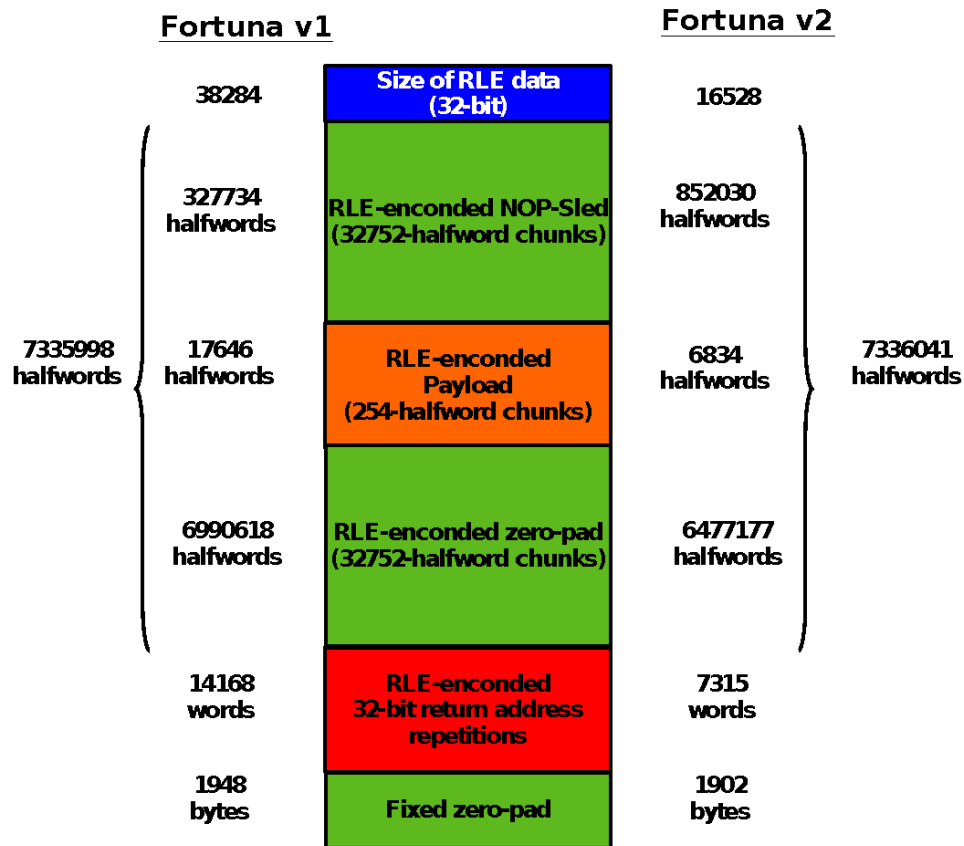
Fortuna is the newest MC-based exploit for PS2 consoles, allowing homebrew software to run on consoles with BIOS v2.30, which are incompatible with FreeMCBoot. As an additional advantage, Fortuna does not require MagicGate Memory Cards (MCs). The vulnerability Fortuna is based on was described first by @TnA some years ago, and developer krat0s exploited it around a year ago. At first, krat0s was willing to prepare a technical write-up about Fortuna's internals. Still, in the end, he decided to maintain this exploit in secrecy due to several reasons I'm not going to describe here.

Fortuna's magic and the secrecy around it made me very curious, so I started reading posts from @TnA, @HWNJ, krat0s, CTurt, and @sp193 in order to get some clues to reverse engineer Fortuna. The conjectures from @TnA were very useful, and they gave some insights; however, @sp193 was the first person that first described some technical details about Fortuna and how it works. His post was the starting point of what I'm about to describe. CTurt's FreeDVDBoot helps me out to polish the exploit implementation as well [1].

How Does Fortuna work?

Fortuna exploits a buffer-overflow vulnerability that can be triggered when the OSDSYS reads an icon (icon.icn), including a compressed texture segment. Compressed texture segments use RLE (Run-Length-Encoding), which is a simple way to zip images. The PS2 icon format and RLE encoding are explained in depth within this document [2], and @sp193 did so for Fortuna RLE encoding on this post [3].

Fortuna uses two icon files, icon.sys and icon.icn. First, icon.sys is a regular file that points to the FORTUNA folder; on the other hand, icon.icn is valid except for the texture segment. Here is where magic lays. From now on, I will explain how Fortuna works from the reverse engineering tasks I performed for the case of the RLE texture segment that is shown in the following figure.



Fortuna v1 and Fortuna v2 were both reverse engineered. They are similar but have slight differences related to the payload and their load addresses. First, Fortuna is based on the exploitation of a buffer overflow vulnerability in OSDSYS. OSDSYS RLE decoder has no boundary check at all. Exploitation is performed by using the following approach: NOP-sled + shellcode + return address repetition + zero pad.

The NOP-Sled technique is used in both versions, whose size is chosen to have a load address for payload (shellcode) of the form: 0x20XX20XX. This is because of the way the RLE decoder repeats the load address by using halfwords. Hence, for Fortuna v1, the load address is 0x20b020b0; and for Fortuna v2, the load address is 0x20c020c0.

Exploit is loaded into RAM when MC contents are shown along with the distinctive Fortuna “white icon.” Once the exploit is loaded into RAM after RLE decoding, it is necessary to trigger payload execution from the load address. This is achieved in the classical buffer overflow exploitation by estimating the offset of the return address on the stack. Fortuna replicates the payload load address to maximize the probability of triggering when the function returns and pops the return address from the stack, which happens when the user hits twice the exit button on the PS2 browser.

After analyzing the compressed texture segment, I found out that for both Fortuna v1 and Fortuna v2 maximum buffer size is 7335998 and 7336041 halfwords, respectively. This is an approximation to the maximum size OSDSYS buffer can have, and this buffer seemingly starts at address 0x20A62050 for the first icon. Further, difference between load addresses for Fortuna v1 and Fortuna v2 exactly corresponds to difference between NOP-Sled sizes $(0xD003E - 0x50036) * 2 = 0x20c020c0 - 0x20b020b0 = 0x100010$.

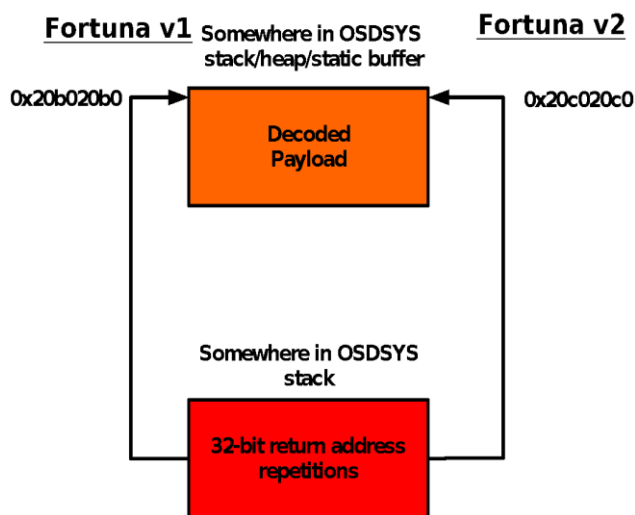
Payload (shellcode) is RLE encoded and injected as chunks of 254 half-words. Fortuna v1 payload is composed of 17646 halfwords, and Fortuna v2 payload is composed of 6834 halfwords. I only disassembled Fortuna v2 payload, which performs the following:

1. The cache is flushed in the usual way.
2. The CPU is placed in kernel mode.
3. The 2nd stage payload is copied beginning from 0x90000 (presumably an ELF loader).
4. Clears some bytes after 2nd stage payload (.bss segment?) (memset from libc is embedded)
5. Flushes cache again
6. And ExecPS2 from 0x900D8.

After the shellcode, the buffer is zero-padded to fill the remaining available space. Zero-pad is composed of 6990618 halfwords for Fortuna v1 and 6477177 halfwords for Fortuna v2. The data below corresponds to the return address repetitions (words) after the payload buffer, which is 14168 for Fortuna v1 and 7315 for Fortuna v2. The number of required repetitions is dependent on the Payload size.

After return address repetitions, there is some zero pad, which is included into RLE encoded data (1948 bytes for Fortuna v1 and 1902 bytes for Fortuna v2). I'm still not sure how OSDSYS RLE decoder interprets that zero-pad; nonetheless, I've been able to trigger the exploit no matter its size.

Finally, after OSDSYS decodes the texture information, the memory layout looks more or less as shown in the next figure.



From this point, the exploit will start running from the load address if return address repetitions were well estimated. To date, I don't know if the payload buffer lays on stack, heap, or if it is a static buffer.

Let's Open Tuna

After figuring out how Fortuna works, the next goal was to create its own icon files with embedded ELF's along with a set of tools to allow users to build their own exploits as needed. In addition, above everything, is to keep software and knowledge free. The above considerations motivated the creation of the OpenTuna project.



OpenTuna uses the same layout of texture segment in Fortuna v1; thus, the load address is 0x20b020b0. OpenTuna uses as a two stages payload:

First stage payload at load address 0x20b020b0 (<https://github.com/parrado/opentuna-payload/tree/main/exploit>)

1. Performs PS2 basic initialization.
2. Launches (ExecPS2) user payload

Second stage payload: Can be any user ELF, compressed or uncompressed. The source code for two sample payload projects is provided along with their corresponding icon files (OPENTUNA-KEYS <https://github.com/parrado/opentuna-payload/tree/main/launcher-keys> and OPENTUNA-NO-KEYS <https://github.com/parrado/opentuna-payload/tree/main/launcher-no-keys>). You're encouraged to use an older SDK to compile payloads.

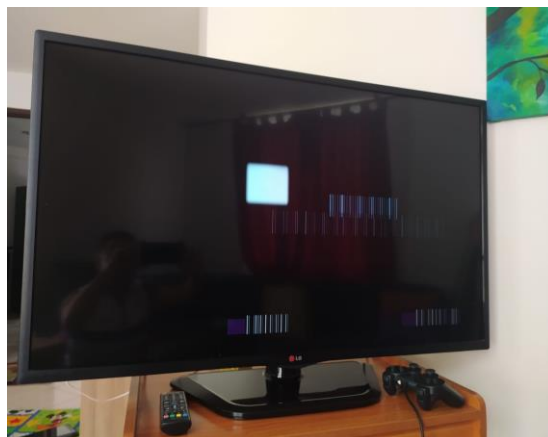
A small utility written in Kotlin will also create the icon file for you from a binary payload <https://github.com/parrado/opentuna-RLE/releases/tag/v1.0> To date, I haven't been able to automate return address repetitions, so maybe you'll be forced to manually hex-edit the generated icon file.

The below figure shows how to perform manual editing of icon.icn to modify return address repetitions to trigger the exploit, in this case for OPENTUNA-KEYS flavor.

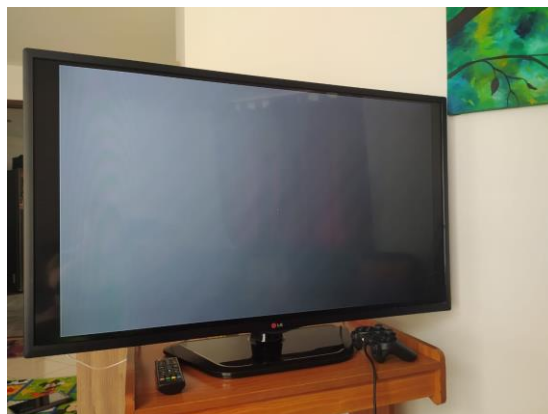
```
00007A40 F0 7F 00 00 F0 7F 00 00 F0 7F 00 00 F0 7F 00 00
00007A50 F0 7F 00 00 F0 7F 00 00 F0 7F 00 00 F0 7F 00 00
00007A60 F0 7F 00 00 F0 7F 00 00 F0 7F 00 00 F0 7F 00 00
00007A70 F0 7F 00 00 F0 7F 00 00 F0 7F 00 00 F0 7F 00 00
00007A80 F0 7F 00 00 F0 7F 00 00 F0 7F 00 00 F0 7F 00 00
00007A90 F0 7F 00 00 F0 7F 00 00 F0 7F 00 00 F0 7F 00 00
00007AA0 B2 42 00 00 F0 7F B0 20 16 77 B0 20 00 00 00 00
```

In this case, return address (0x20b020b0 in Little Endian encoding) is repeated $(0x7ff0+0x7716)/2=0x7b83$ times.

You should test your exploits on actual hardware as PCSX2 tends to give false positives. To adjust return address repetitions, you can follow the next procedure. Use as a template a sample icon file, and if you watch a messed OSDSYS browser like the one in the following Figure, you probably are missing return address repetitions, so increase them.



Conversely, if you completely freeze the console when loading MC contents as shown in the next Figure, you indeed include too many repetitions, so reduce them.



Finally, keep in mind that one chunk of repetitions cannot be larger than 0x7fff; if you need more repetitions, please use the next available word (32-bit).

Bibliography

- [1] “FreeDVDBoot - Hacking the PlayStation 2 through its DVD player.” [Online]. Available: <https://cturt.github.io/freedvdboot.html>. [Accessed: 16-Feb-2021].
- [2] “(No Title).” [Online]. Available: <https://www.ps2savetools.com/ps2icon-0.5.pdf>. [Accessed: 16-Feb-2021].
- [3] “PS2 - Fortuna. ALL PS2s (incl. TV) HACKABLE! Another discless exploit! | Page 5 | PSX-Place.” [Online]. Available: <https://www.psx-place.com/threads/fortuna-all-ps2s-incl-tv-hackable-another-discless-exploit.27226/page-5#post-221178>. [Accessed: 16-Feb-2021].